MT: A Toolset for Specifying and Analyzing Real-Time Systems

P. C. Clements, C. L. Heitmeyer, B. G. Labaw, A. T. Rose

Center for High Assurance Computing Systems
U.S. Naval Research Laboratory
Washington, DC 20375

Abstract

This paper introduces MT, a collection of integrated tools for specifying and analyzing real-time systems using the Modechart language. The toolset includes facilities for creating and editing Modechart specifications. Users may symbolically execute the specifications with an automatic simulation tool to make sure that the specified behavior is what was intended. They may also invoke a verifier that uses model-checking to determine whether the specifications imply (satisfy) any of a broad class of safety assertions. To illustrate the toolset's capabilities as well as several issues that arise when formal methods are applied to real-world systems, the paper includes specifications and analysis procedures for a software component taken from an actual Navy real-time system.

1. Introduction

During the past decade, software engineering researchers have devoted increased attention to *formal methods* --mathematical techniques useful in software development [10]. Recently, an international survey was conducted of 12 real-world systems whose development involved formal methods [7]. The 12 systems spanned a wide range of applications, from safety-critical software, such as the FAA's TCAS, a collision avoidance system for commercial aircraft [19]; to secure systems, such as Multinet Gateway, a protocol-based service for secure datagram delivery [8]; to hardware developments, such as the INMOS Transputer, a family of 32-bit VLSI circuits [2]. The survey produced a number of important conclusions about the nature of formal methods usage in current software development [7]. Three of the conclusions are relevant to this paper:

- A number of formal methods for representing and reasoning about behavioral requirements (e.g., the Z language [21]) were found to have significant utility in the development of real-world systems.
- Solid tool support for formal methods is practically nonexistent. The few tools available are weak and lack robustness.
- Formal methods for representing and reasoning about timing requirements are lacking. Although researchers have introduced a number of formal methods for specifying and analyzing timing behavior, these have not yet been used on real applications.

One reason that real-time formal methods are not being used is that *how* they should be used is still unknown. Moreover, although there is widespread agreement that tools supporting the methods are needed, what form these tools should take is also unknown. The purpose of this paper is to introduce MT, a prototype development environment that addresses these issues. MT's goals are to provide an assessment of some of the more promising real-time methods, to determine what tools are needed to support the methods, and to explore how these tools can be used together to develop real-time systems.

MT provides comprehensive support for applying formal methods to the design and construction of real-time systems. The focus of MT is on *hard real-time systems*--systems that must, without exception, deliver results within specified time intervals. MT supports the *formal specification* of real-time behavior in a language called Modechart [15] and *formal analysis* via formal verification, simulation, and completeness and consistency checking. The objective of formal analysis is to improve the correctness of the Modechart specifications.

This paper introduces the tools that make up the MT toolset[1] [12] and shows how the tools can be used to specify and analyze the behavior of a scheduler, a representative example whose origins are in a real-world avionics system [4]. The full specification of this example is presented in [5].

2. Formal methods for real-time systems

Several studies have shown that errors in specifications are the most frequent types of software errors and the most expensive to correct [3] [9]. By removing imprecision and ambiguity and by reducing incompleteness and inconsis-

[1]MT is implemented in C and C++ for the UNIX/X-windows environment. Its run-time code is approximately 7.8MB.

1052-8725/93 $03.00 © 1993 IEEE

tency, formal methods can significantly reduce the number of errors in specifications. Unlike informal approaches to specifications, such as natural language descriptions, expression of a software component's requirements in a formal language produces a precise, unambiguous statement of what is needed. Also, in contrast to informal approaches, formal methods can organize the specification so that instances of inconsistency and incompleteness are easier to detect.

An additional benefit of formal specifications is their amenability to formal analysis. One form this analysis can take is *formal verification*, which checks formal specifications for critical application properties. In addition, the formal specifications can drive simulations of the system. By running a series of simulations, each representing one possible execution of the system, the user can determine whether the system behavior represented by the specifications is consistent with his intent. Thus, simulation supports the *validation* of the specifications.

The formal method that underlies MT is the Modechart language [15], a graphical language based on concurrent finite state diagrams, such as those used in Statechart [11], and the concept of modes introduced by references [14] [1] to simplify software requirements specification. Modechart has a formal semantics defined by Real-Time Logic (RTL), a form of first-order logic [17]. Fundamental constructs of Modechart are modes, mode transitions, actions, events, and timing constraints. *Modes* describe control information that imposes structure on a system's operation. An *action*, one of which may be associated with each mode, is an operation that is executed when a mode becomes active. An *event* refers to a moment in time when something occurs, such as a mode transition, a discrete change in the system's environment, the start or stop of an action, or a variable taking on a new value. To specify timing constraints on mode transitions, Modechart offers *delays* (lower bounds on the time interval from mode entry to mode exit) and *deadlines* (upper bounds).

MT includes a mechanical verifier [18] [22] developed at the University of Texas, one of the few mechanical verifiers available for reasoning about real-time systems [13]. The purpose of the verifier is to determine whether a timing or behavioral assertion (expressed in RTL) is logically implied by a set of Modechart specifications. Each assertion is a formal, logical statement of a property that must hold for the specifications to be considered correct.

3. Introduction to the toolset and the demonstration problem

3.1 The application

To demonstrate the formal methods, we have focused on the control system of a tactical air-launched missile that detects ground targets based on their electromagnetic emissions. In specifying the system, we have made some simplifying assumptions [5]. The system is representative of a modern, parallel, embedded system with critical performance requirements. As such, it is a good vehicle for our experiments. The missile software is written in a high-level programming language (Pascal) and uses a real-time operating system. The operating system provides a scheduler and primitives that allow synchronization of fixed-priority, preemptive tasks. Mailboxes provide the mechanism for task communication. A task that requests the CPU may need to wait if certain conditions apply. For example, writing to a full mailbox may cause a task to wait until an item is removed from the mailbox; reading an empty mailbox may suspend a task until an item is added to the mailbox. The system runs on several processors operating in parallel; a copy of the operating system resides on each processor.

In this application, we are using Modechart and the MT environment to aid in *design validation*. A design for the missile software has already been proposed (and in fact, implemented). We are using MT to specify and analyze the design to make sure that it satisfies chosen timing and behavioral properties.

For demonstration purposes, we have selected a six-task subset called *Process_Signals* for our study. This subset represents the most complex part of the application and provides a number of modeling challenges--parallel processing, task synchronization, pipeline processing, priority-based dynamic scheduling with task suspension, data-dependent computation--that should fruitfully test real-time technology. *Process_Signals* analyzes the signals received by the missile's sensors and produces an azimuth/elevation pair that is transmitted to the missile's guidance system. It uses a pipeline approach to target acquisition and tracking, thus allowing two or more sets of sensor data to be processed simultaneously.

Figure 1 illustrates our top-level Modechart decomposition of the missile system. In the decomposition, the system is described by four concurrent subsystems--the environment, the operating system made up of the scheduler and the task synchronization mechanism, the *Process_Signals* subset, and the tasks outside of *Process_Signals*. In Figure 1, the letters 'P' and 'S' denote modes whose children operate in parallel or in series with each other, respectively. The environment consists of two concurrent subsystems: one signals environmental events to which *Process_Signals* must react, a second removes from the environment's mailbox events sent by *Process_Signals*. The operating system consists

of two parts: one schedules the tasks based on task priority, and the other provides the task synchronization (mailbox) mechanisms.

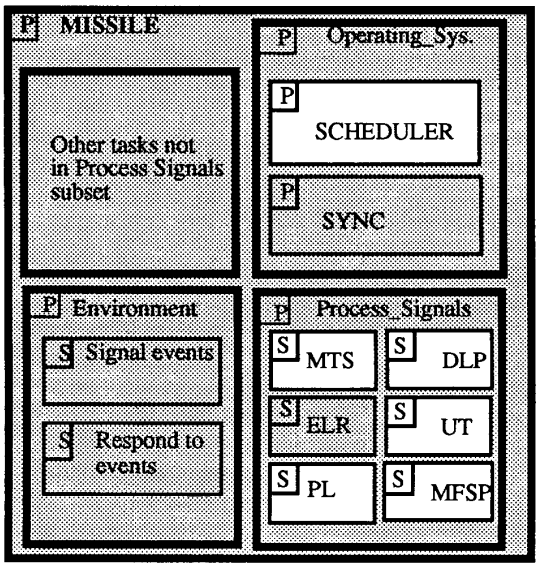

FIGURE 1. Highest-level Modechart model of the missile system.

The remaining discussion focuses on a small subset of the Modechart specification shown in Figure 1, namely, the scheduler for one of the CPUs in the missile and the five application tasks that run on it. These modes are shown in white in the figure.

Figure 2 illustrates the structure of each of the five *Process_Signals* tasks. To model processor allocation, we introduce communication between the modes modeling the tasks and the mode modeling the scheduler as follows: A task T is modeled by a mode called T. If T wishes to acquire the processor, a submode T.wait is entered. Until that time, T.init is active. If T is awarded the processor while in T.wait, it enters T.work. If T is preempted before it is through, the mode transitions from T.work back to T.wait. If the task completes its work, the mode transitions to T.done, and then back to T.init where it waits for the next time to be activated. Task T is awarded the CPU, and may begin work, at the moment that the Scheduler submode called T.awarded_cpu is entered. The scheduler chooses a task based on priorities among all tasks whose modes are currently in their wait state. One time unit after entering the appropriate T_awarded_cpu mode, the Scheduler transitions to T_has_cpu, where it remains until T is preempted or relinquishes the cpu[2].

In our example, each task requests the CPU a fixed amount of time after it last finishes its work; the time varies for each task. Any modeling of the task's functionality would occur inside T.work, but since we are only modeling processor allocation, the work mode is empty. A task voluntarily relinquishes the CPU when it finishes its work, which occurs after a fixed interval of uninterrupted CPU time. A task that is preempted before completion starts from the beginning when it next gets the CPU; this models "start-over" pre-emption, as opposed to the more usual resumptive pre-emption.

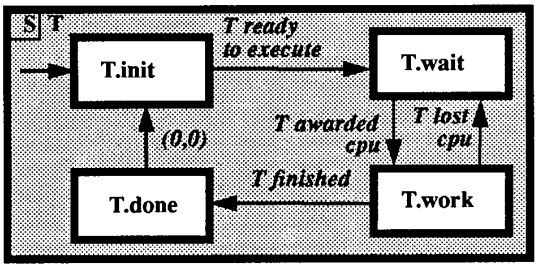

FIGURE 2. Internal structure of a mode that models a task. Entry into the "wait" mode constitutes a request to the scheduler for the CPU.

3.2 Creating and editing specifications

MT is a window-based toolset that supports the creation, editing, and layout of a set of Modechart specifications. Figures 3 and 4 show two windows that appear when the user names a set of specifications he wishes to edit. The window in Figure 3, the Locator Window, always displays the complete Modechart specification. The user selects the portion of the specification he wishes to work on by moving and resizing the bold rectangle until it covers the desired area. The portion selected is presented for editing in the Work Window, shown in Figure 4. Initial children of serial modes are outlined with thicker lines than other modes. In Figure 4, CPU1_idle is an initial mode.

MT supports a direct-manipulation style user interface for creating, moving, and deleting modes and mode transitions and for resizing modes. It has several advanced features, such as zooming, Undo, and Redo operations; an object finder, which centers the display on a named object; and a program that improves the layout of the specifications. The results of applying the layout program to the specifications for mode CPU1 appear in the Work Window shown in Figure 4.

[2]The T_has_cpu modes exist to avoid a *zero cycle*, a loop in which an infinite number of transitions are required to be taken at the same time instant. Zero cycles are illegal in Modechart specifications.

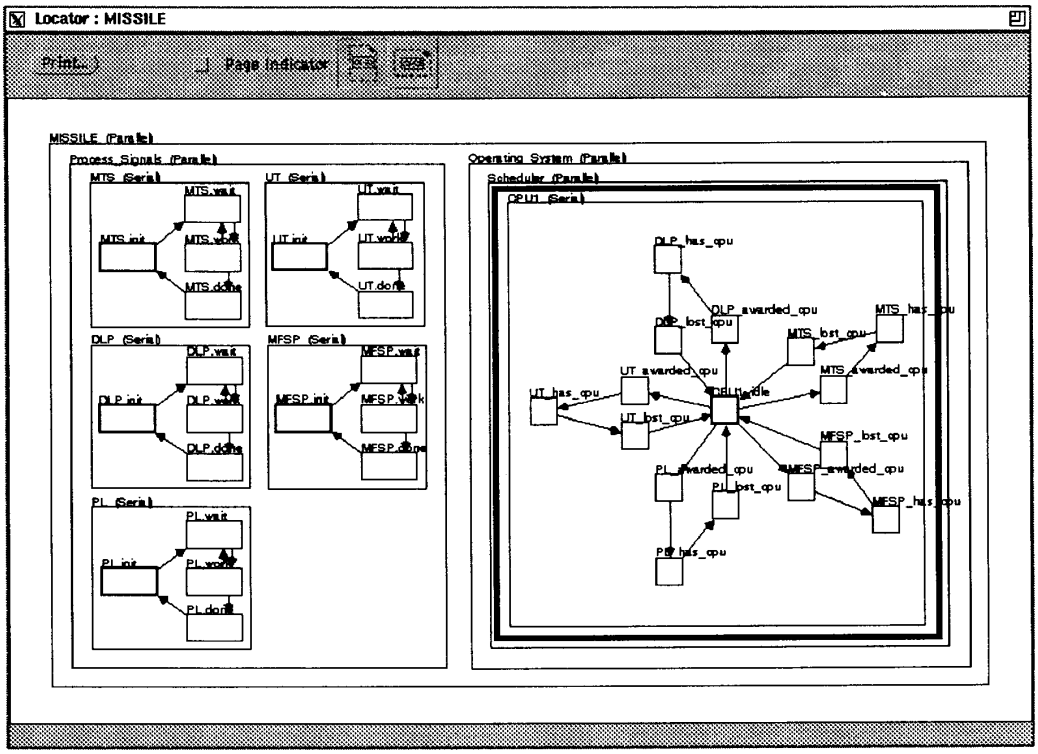

FIGURE 3. MT's Locator Window. Users move and resize the bold rectangle to determine the area displayed in the Work Window, shown in Figure 4.

3.3 Static consistency and completeness checking

The purpose of static consistency and completeness checking is to detect possible errors in the specifications prior to simulation and verification. Unlike the simulator and the verifier which deal with the run-time behavior implied by the specifications, MT's Consistency and Completeness Checker performs a static analysis. Using syntactic information only, the Checker searches the specifications for unreachable modes, sink modes (modes from which there is no exit), and modes with self-transitions. It also looks for statically nondeterministic transitions[3]. Although the Modechart semantics permit some of these anomalies (e.g., nondeterministic transitions and sink modes), the Checker notifies users of their existence to make sure they are intentional and not errors.

[3]If the system is in serial mode M and an event causes the system to exit from one of M's children, a statically nondeterministic transition exists if there exist transitions to any one of two other modes, both of which are also M's children, that share the same trigger. This is a static check, because the transition definitions are compared textually. The verifier, discussed later, provides a test of dynamic nondeterminism, in which two transitions from the same mode are both eligible to be taken at the same time because (perhaps different) events happened to occur simultaneously.

3.4 Simulation

Once a chart has been entered and checked for consistency, the tool's simulator can be invoked [23]. To run the simulator, the user first assembles a file of options (or uses a default set) that tells the simulation tool how nondeterministic decisions are to be made. These decisions include choosing a transition to take when more than one is enabled, scheduling the next time a randomly-occurring external event is to occur, and fixing the execution time of an action between given bounds. For example, although external events may occur any time after a specified delay (also referred to as a minimum separation), the user may instruct the simulator to model these events at specific times, at random times with given statistical distributions, or never. The user also specifies how long he wishes the simulation to run, the set of objects (e.g., modes) to be displayed, and a set of breakpoints. The simulator uses breakpoints much like a symbolic debugger. When a breakpoint occurs, the simulator halts to allow the user to inspect or to alter the computation. Breakpoints may occur at specified times or time multiples or they may coincide with events, e.g., whenever a particular mode is entered.

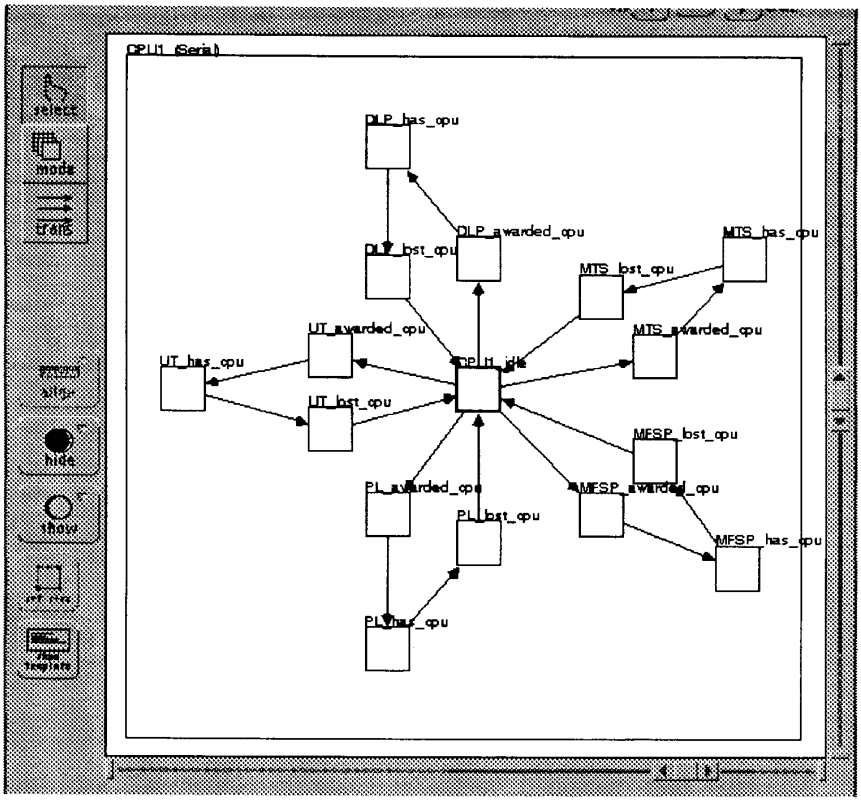

FIGURE 4. MT's Work Window, showing that portion of the specification designated by the Locator Window in Figure 3.

The simulator produces a bar graph, as shown in Figure 5. Each bar represents the behavior of a single Modechart object over time, where time begins at zero and is displayed horizontally from left to right. For modes, a thick line indicates that the mode is active at a particular time; a thin line indicates that it is not active. The simulator can also display the temporal behavior of external events, actions, transitions, and boolean variables.

Using the specification shown in Figure 3, Figure 5 shows the simulation output for the scheduler and three of its five application tasks. The display has been ordered so that the modes relevant to each task are grouped together for easy reference. The task groups are ordered in decreasing priority. Not all modes of the specification are displayed.

From this display we can observe the following behavior, which was expected and gives us some preliminary confidence in our specification:

- Modes CPU1, MTS, UT, and MFSP are always active. This is expected, because each is a child of the root mode, which is parallel. Their graphs are displayed in the simulation not because their behavior is interesting -- the behavior of their *children* is what we care about -- but because the solid lines serve as convenient visual delimiters for the scheduler and task groups.
- Each task is awarded the CPU only *after* it enters its wait state, and never before. No task is awarded the CPU unless it is in the wait state.
- No two tasks are awarded the CPU simultaneously; no two tasks are in their work states simultaneously.
- No task is in its work mode after a higher-priority task enters its wait mode.
- MTS, the highest priority task, never has to wait for the CPU longer than the tick necessary for the CPU to switch to a new task. Other lower priority tasks sometimes must wait a long time in their wait states before they are awarded the CPU and can enter their work states. For example, task UT requested the CPU at time 82, but did not get it until time 97 because higher-priority tasks (not shown) were using it.

- We observe task pre-emption occurring at time 20. UT, the fourth-highest priority task, was awarded the CPU at time 16. However, at time 20, a higher-priority task (not shown) requested it, and UT was preempted. Task UT did not receive the CPU again until time 48; in the meantime, the CPU was busy servicing other tasks. Task UT retreated to mode UT.wait for this interval waiting for its next turn with the CPU.
- All of the tasks appear to receive the CPU long enough to complete their work, except for MFSP.

All of these observations, and many more, concern aspects of the model's behavior that we hope are true and invariant. Although we cannot conclude that the specification is correct with respect to these invariants, seeing them satisfied in this execution path does increase our confidence in the specification. In this example, the simulation also suggests the possibility of incorrect behavior.

In particular, task MFSP appears to be subject to starvation, since it does not appear to have the CPU for enough uninterrupted time to finish its work. In the interval of this simulation, it enters the work mode at time 8 (having requested the CPU before any other task) but is soon preempted and forced to retreat back to its wait mode. For the rest of the shown duration, MFSP never resumes execution for more than a single time unit.

Finally, we can observe the following, which was neither expected nor unexpected. That is, after an initial period in which no task requests the CPU, the CPU is never idle (at least during the simulation interval). In Figure 3, the ticks on the CPU1_idle mode line indicate zero-duration activations of the mode. These occur when the task that was running relinquishes the CPU to another task.

In general, simulation is to specification what testing is to software. Because it only exercises one execution path per run, the simulator can only show the presence of errors and not their absence. In spite of this limitation, the simulator has proven valuable, because it provides a visual representation of an execution path. With this approach, unexpected behavior is quickly identified. The errors detected can be used to help identify invariants about the specification (namely, that the error state does not occur) that the user can later try to confirm with the verifier.

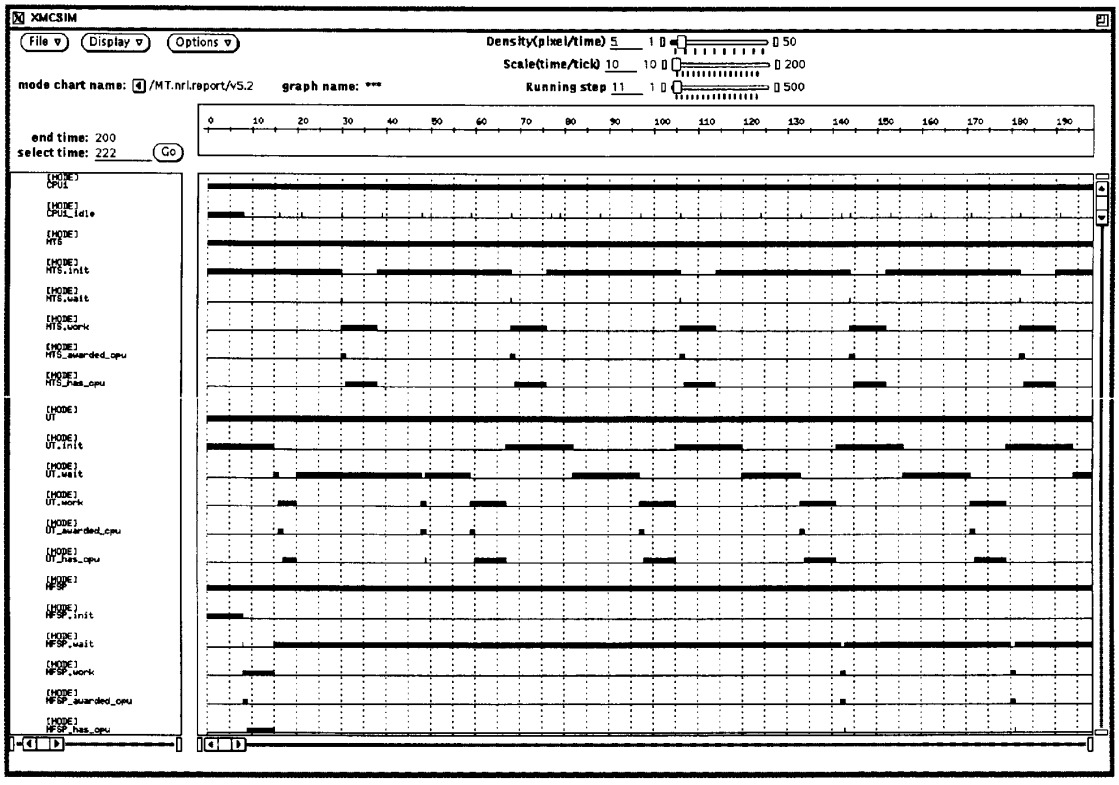

FIGURE 5. The MT Simulator window.

3.5 Verification

The MT verifier [16][18] either determines the validity of a given predicate or answers specific queries about the specifications, such as whether selected modes are reachable. To perform these tasks, the verifier derives a computation graph from the Modechart specifications. The nodes of the graph represent all possible states of the system; the edges are labeled with events that can cause state changes.

Figure 6 shows and explains the verifier's main work window. The Command and Write buttons access menus that allow the user to respectively perform analysis on and report information about the specification. We illustrate some of the features by asking questions about the scheduler example raised during simulation.

The commands are presented in the form of RTL formula templates [15] with blanks in strategic places. The user chooses specification objects, constants, and relations to fill in the blanks. When the formula is complete, "Evaluate" will return true or false. The user can also evaluate an incomplete formula by asking the verifier to fill in the integer constants, if any, which will cause the formula to be true; this is called "Calculate Offsets".

Because of restrictions in the current version of the verifier tool, the modechart used to demonstrate the verifier is topologically different (but semantically equivalent) to the one presented in Figure 3.1. See [5] for more details.

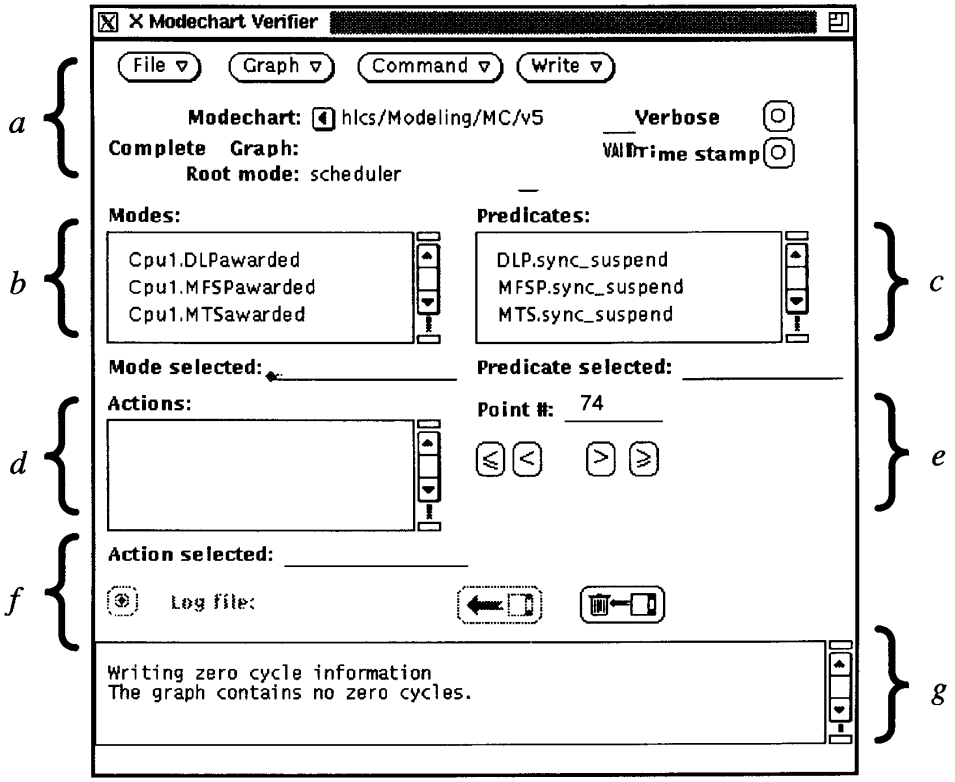

FIGURE 6. The MT verifier's main work window. Area *a* shows information about the Modechart specifications and its computation graph. In particular, it shows that the graph is valid (has passed internal consistency checks). Areas *b-d* list objects in the specification (modes, state variables, and actions, respectively). The user can select from these lists to build verification queries. (This example contains no actions.) Area *e* is for selecting points in the computation graph about which to construct queries. The buttons labelled with relation signs allow the user to step through valid points numbers of the graph. Area *f* controls logging to a file, and *g* is the output window.

Does MTS, the highest-priority task, get the CPU whenever it wants it? In our model, MTS asks for the CPU 30 time units after it exits its initial state. The verifier's *separation* query determines the minimum or maximum possible time separation between two consecutive occurrences of a particular event over all computations. Figure 7 illustrates the separator query which shows that MTS does get the CPU whenever it requests it. Minimum or maximum is selected by switching the relation symbol using the "↑↓" button. Here, we've shown that the time between subsequent entries into mode MTS.work is at most 41 time units; the "41" was supplied by the tool in response to our "Calculate Offset" request. Since MTS.work completes in 8 time units, 2 time units are devoted to task switching, and the relation is "greater than" as opposed to "greater than or equal to", 41 (30 + 8 + 2 + 1) is what we expect.

Does task MFSP starve? In simulation, MFSP appeared to starve because we never saw an activation interval for MFSP.work lasting the requisite time to complete its work. The verifier's *elapsed time* query can tell us the minimum and maximum activation interval for a mode, over all computations. Figure 8 shows that the least amount of time that MFSP.work is active is one time unit, and that the most it is active is seven time units. Since a task requires eight time units to complete its work, this confirms that MFSP starves under all computation trajectories.

Are two tasks' work modes active simultaneously? The verifier's *reachability* query allows the user to describe states in the computation and reports how many points in the computation graph satisfy that description. States are described by specifying which modes are and are not active in that state, and which state predicates are true and which are false. Figure 9 illustrates a reachability query for states in which both MTS.work and PL.work are active. The verifier reports that there are zero such points in the graph, as expected. Reachability queries about all pairs of work modes can confirm that no two tasks share the CPU simultaneously for a non-zero time interval.

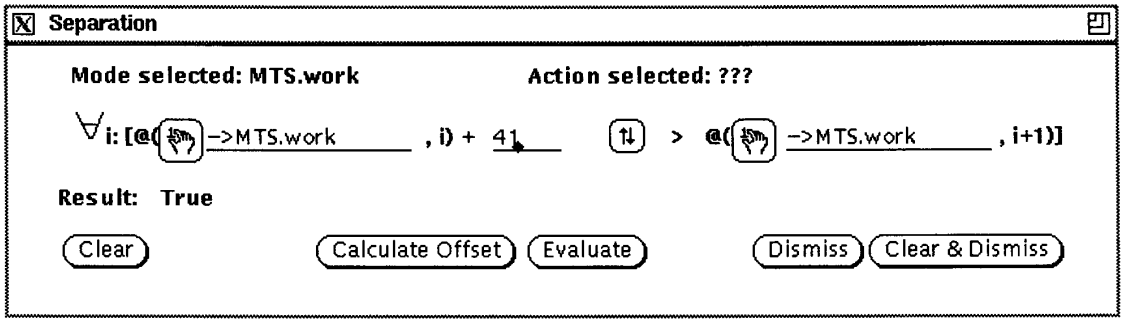

FIGURE 7. The MT verifier's separation query.

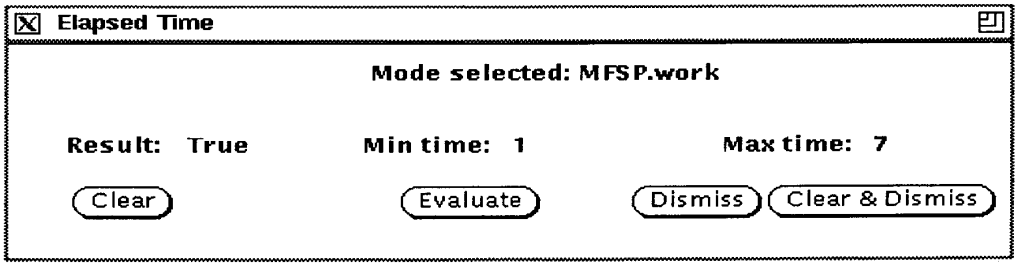

FIGURE 8. The MT verifier's elapsed time query.

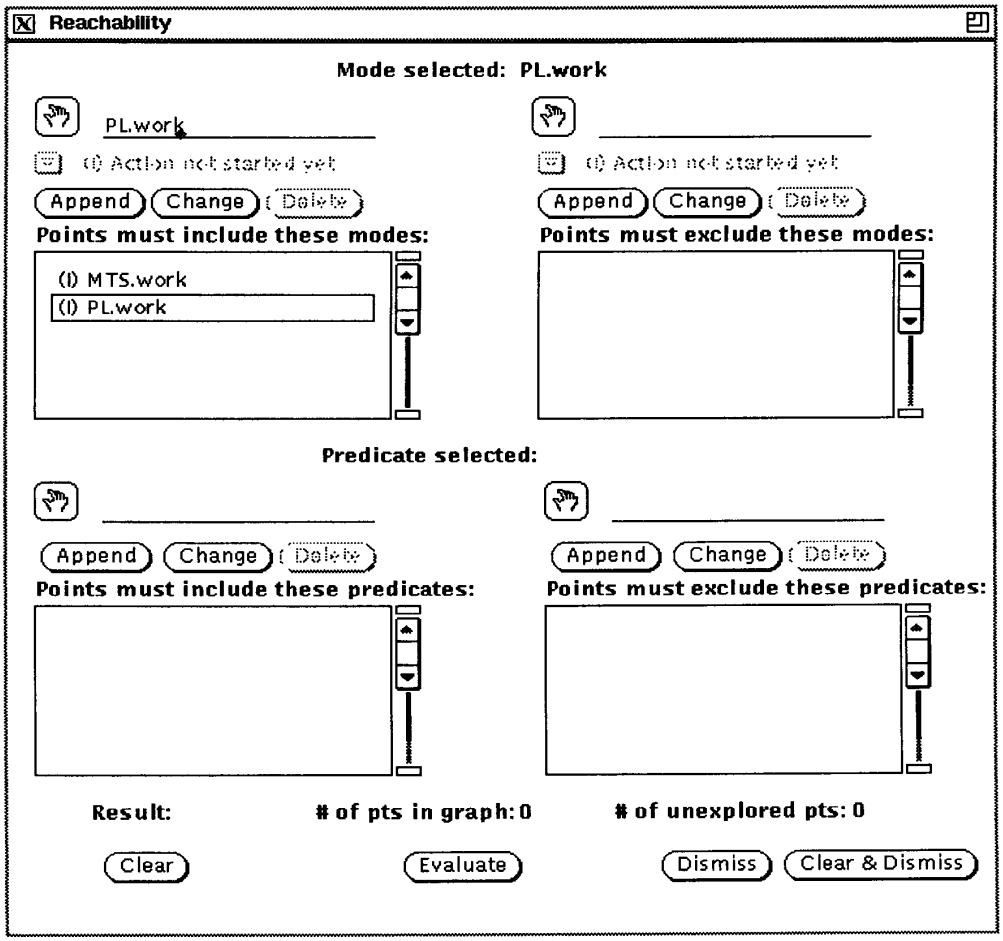

FIGURE 9. The MT verifier's reachability query. The four components are for specifying, respectively, a list of active modes, a list of inactive modes, a list of true predicates, and a list of false predicates. The query returns the number of points in the computation matching the parameters. Objects not mentioned in the lists assume a "don't care" status in the query.

Does every work state follow an awarding of the CPU? The verifier's *all-universal* query, which is shown in Figure 10, is designed to test assertions about events that should be strongly synchronized with each other. It returns the minimum separation between corresponding occurrences of pairs of events. The query shown verifies that the i^{th} entry into DLP's work state always follows the i^{th} entry into the scheduler mode that awards the CPU to DLP, perhaps by as little as zero time units. The fact that there is an offset (here, zero) for which this lockstep formula holds means that the DLP task never goes to work without permission.

Other queries can verify this safety property for the other tasks. This query can operate on a list of related event pairs, and evaluates the conjunction of the individual separation formulas.

Is a task ever given the CPU without requesting it? Substituting the appropriate wait and awarded_cpu modes in the *all-universal* query in Figure 10 would show whether a task is awarded the CPU only in response to a request for it.

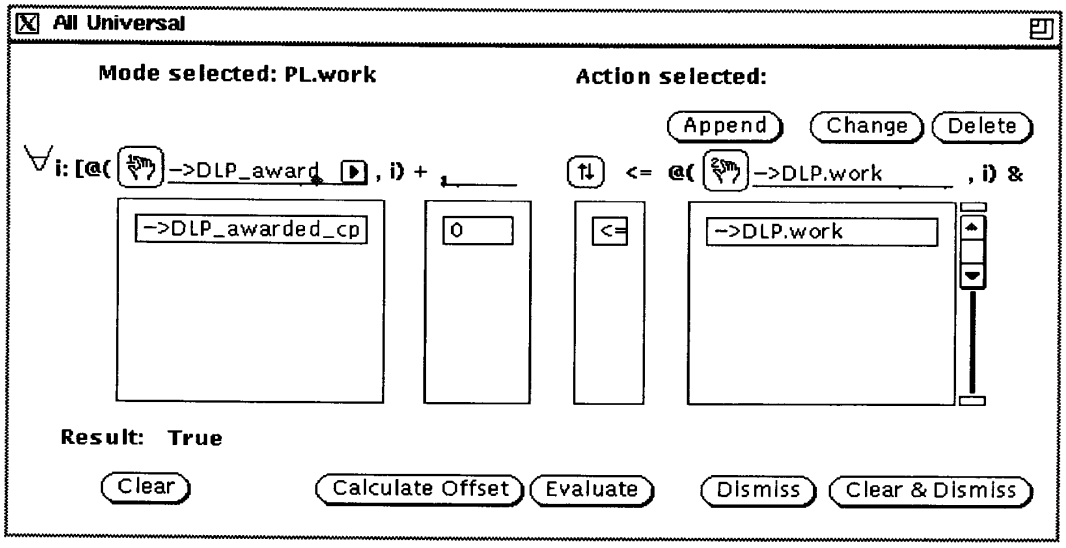

FIGURE 10. The MT verifier's all-universal query.

4. Conclusions

MT has several important attributes.

- Unlike other real-time tools, MT provides a *comprehensive, unified* approach to specifying real-time systems and analyzing their behavior via simulation and mechanical verification.
- The MT tools have a common formal semantics defined by Real-Time Logic.
- MT is one of the few toolsets in existence that produces graphical specifications that are both easy to understand and formal. The specifications form the basis for formal analysis.

Technology for specifications that support automated analysis of real-time systems is in its infancy. If it is to grow beyond this stage, progress must be made in the following areas:

- Verification based on model-checking suffers from a state explosion problem. It is easy to create a specification for which verification is not practical on even the largest machines. Ways must be found to structure specifications into subsets, so that the properties we wish to verify about each subset are provably independent of the other subsets, except in very limited and well-managed ways. There are also ways to shrink the computation graph by exploiting information about the specification (particularly where it describes, or is intended to describe, deterministic behavior). New approaches are needed.
- There are several user interface issues that must be addressed. Are there better ways to present the modecharts to the user? What is the best way to provide replication (copy-and-paste) facilities so that the user can quickly build a large modechart from almost-alike components? Can we distance the user more from the computation graph and less from the Modechart specification during verification? How can the verifier help the user determine *why* a query failed, as opposed to just reporting that it did?

An interim report is in progress that contains a large Modechart specification for the missile example [6]. The report presents the specification at a high level and concentrates on the design and modeling decisions that were made to achieve a specification amenable to the planned analysis. These decisions were almost always ones that helped us to manage complexity, and they are of the type that go unrecognized until a formal method is used to specify something other than a small example.

A second interim report presenting a precise formal model of the application is also in progress. This report will provide detailed information so that others can apply their own specification, verification, and other formal methods technology to the same problem. The objective is to provide an authentic example of a real-time problem so that different formal methods may be compared.

Acknowledgments

The authors acknowledge the long hours and dedicated work of the many who helped develop the toolset. A. Bull, C. Gasarch, and M. Pérez of NRL made substantial contributions to the toolset's facilities for creating and editing specifications and to the design of the graphical user interface. Doug Stuart of the University of Texas built the underlying verifier and modified it to work with the graphical user interface. Alex Ho of UT designed and implemented both the verifier's user interface and the user interface for the Modechart simulator. Al Mok of UT proposed the barchart display used by the simulator.

Bibliography

[1] T. A. Alspaugh, S. R. Faulk, K. H. Britton, R. A. Parker, D. L. Parnas, and J. E. Shore, "Software Requirements for the A-7E Aircraft," NRL Rep. 9194, Naval Research Lab., Wash., DC, 1992.

[2] G. Barrett, "Formal Methods Applied to a Floating Point Number System," *IEEE Trans. Softw. Eng.* SE-15, 1989, 611-621.

[3] B. Boehm, *Software Engineering Economics*, Englewood Cliffs, NJ, Prentice-Hall, 1981.

[4] P. C. Clements, C. L. Heitmeyer, B. G. Labaw, A. K. Mok, "Applying Formal Methods to An Embedded Real-Time Avionics System," *Proc., IEEE Real-Time Applications Workshop*, New York, NY, May 11-12, 1993.

[5] P. C. Clements, C. L. Heitmeyer, B. G. Labaw, A. T. Rose, "A Toolset for Specifying and Analyzing Real-Time Systems: Overview and Example," NRL Rep. 7405, Naval Research Lab., Wash., DC, 1993.

[6] P. C. Clements, A. Bull, and B. Labaw, "Modeling the HARM Low-Cost Seeker System with Modechart," NRL report (in preparation).

[7] D. Craigen, S. Gerhart, and T. Ralston, "An International Survey of Industrial Applications of Formal Methods," NRL Report 9581/9582, Naval Research Lab., Wash., DC, 1993.

[8] G. Dinolt et al., "Multinet Gateway---Towards A1 Certification," *Proc., IEEE Symp. on Security and Privacy*, 1984.

[9] R. Fairley, *Software Engineering Concepts*, New York, NY, McGraw-Hill, 1985.

[10] *IEEE Software*, Sep. 1990, Special Issue on Formal Methods.

[11] D. Harel et al., "Statemate: A Working Environment for the Development of Complex Reactive Systems," *IEEE Trans. Softw. Eng.* SE-16, 4, Apr. 1990.

[12] C. L. Heitmeyer, P. C. Clements, B. G. Labaw, A. K. Mok, "Engineering CASE Tools to Support Formal Methods for Real-Time Software Development," *Proc., CASE '92 Fifth Intern. Workshop on Computer-Aided Softw. Eng.*, Montreal, Canada, July 6-10, 1992.

[13] C. L. Heitmeyer and B. G. Labaw, "Requirements Specification of Hard Real-Time Systems: Experience with a Language and a Verifier," in *Foundations of Real-Time Computing: Formal Specifications and Methods*, A. van Tilborg and G. Koob, Eds., Kluwer Academic Publishers, Norwell, MA, 1991, 291-313.

[14] Heninger, K.L., "Specifying Software Requirements for Complex Systems: New Techniques and their Application," *IEEE Transactions on Software Engineering*, vol. SE-6, no. 1, pp. 213, January 1980.

[15] F. Jahanian, R. S. Lee, and A. K. Mok, "Semantics of Modechart in Real Time Logic," *Proc., 21st Hawaii International Conference on System Sciences*, January 1988.

[16] F. Jahanian and D. A. Stuart, "A Method for Verifying Properties of Modechart Specifications," *Proc., Real-Time Systems Symposium*, Orlando, FL, Dec. 1988.

[17] F. Jahanian and A. K. Mok, "Safety Analysis of Timing Properties in Real-Time Systems," *IEEE Trans. Softw. Eng.* SE-12, 9, Sep. 1986, 890-904.

[18] F. Jahanian and D.A. Stuart, "A Method for Verifying Properties of Modechart Specifications," *Proc., Real-Time Systems Symposium*, Huntsville, AL, Dec., 1988.

[19] N. Leveson et al., "Experiences Using Statecharts for a System Requirements Specification," *Proc., Intern. Workshop on Software Specif. and Design*, Como, Italy, Oct. 25-26, 1991, pp. 31-41.

[20] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053-1058, 1972.

[21] J. M. Spivey, *Understanding Z: A Specification Language and its Formal Semantics*, Cambridge Univ. Press, 1988.

[22] D. A. Stuart, "Implementing a Verifier for Real-Time Systems," *Proc., Real-Time Systems Symposium*, Orlando, FL, Dec. 1990, 62-71.

[23] D. A. Stuart and P. C. Clements, "Clairvoyance, Capricious Timing Faults, Causality, and Real-Time Specifications," *Proc., Real-Time Systems Symposium*, San Antonio, TX, Dec. 3-6, 1991.